МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Рекурсивная обработка иерархических списков

Вариант 22

Студент гр. 8304 ___ Нам Ё Себ

Преподаватель ___ Фиалковский М. С.

Санкт-Петербург

2019

Цель работы.

Получить опыт работы с рекурсивной обработкой иерархических списков.

Постановка задачи.

Алгебраическое (+, -, *, sqrt(), log(,)), проверка синтаксической корректности, простая проверка log(,), префиксная форма.

Описание алгоритма.

1) Считывание осуществляется при помощи конструкции:

```
while (std::getline(in, currentFileString))
```

2) После считывания очередной строки, осуществляется замена конструкций вида sqrt(), log(,) на 0. Замена на 0 объясняется тем, что нет необходимости вычислять значение выражения, нужно только проверить его синтаксическую корректность.

3) Далее осуществляется рекурсивная проверка ранее считанных строк и построение иерархического списка.

Спецификация программы.

Программа предназначена для проверки строк, хранящихся в файле F, и записи результата проверки в файл G. Программа написана на языке C++.

Описание функций и структур данных.

1) Для замены конструкций вида sqrt() и log(,) были реализованы функции convertSqrtToInt и convertLogToInt, которые итеративно заменяют каждую вышеупомянутую конструкцию на 0. Поиск данных конструкций осуществляется при помощи регулярных выражений. Так же реализованные регулярные выражения написаны без учета отрицательных аргументов функций sqrt и log, следовательно, они не будут заменены на 0 и дальнейшая проверка уже измененной строки будет ложна, что и необходимо.

2) Для решения поставленной подзадачи была реализована функция check, которая осуществляет рекурсивную проверку ранее измененной строки. Рекурсивно обрабатываются скобочные конструкции в исходном выражении. Также данная функция формирует иерархический список, каждый элемент которого является объектом класса Node. Данный класс содержит 2 поля – поле arguments, которое является объектом класса std::variant, что позволяет обрабатывать случай с бинарной или унарной операцией, и поле value, которое аналогично первому полю является объектом класса std::variant, данное поле содержит либо операцию (для типа char), либо операнд (для типа int).

Тестирование.

Таблица 1 – Результаты тестирования программы

Содержимое файла F	**Содержимое файла G (после запуска программы)**
(+ a (+ b c)) ((a 3)(b 2)(c 3))	YES
(+ a b ((a 2)(b 3))	Uncorrect brackets statement
+ a b) ((a 2)(b 3))	Uncorrect brackets statement
lekngowogniwoginwogn ()	NO
(+ sqrt(a) (+ log(a, 3) log(3, b))) ((a 2)(b 3))	YES
(+ sqrt(a) (+ b c)) ((a 3)(b 2))	NO
(+ a (+ b)) ((a 3)(b 2))	NO
/\/\/\/\/\/\/\/\/\/\/\/\ ()	NO
(* (- log(a, -2)) (- b c)) ((a -3)(b 0)(c -0))	NO
(* (+ (- (- a) (+ b c)) +2) (- a b)) ((a -3)(b -1)(c -5))	YES

(- log(abc, cba) (+ cba bac)) ((abc 3)(cba -1)(bac -2))	YES

Выводы.

В ходе работы был получен опыт работы с рекурсивной обработкой иерархического списка. Исходный код программы представлен в приложении А.

Приложение А. Исходный код программы.

Main.cpp

```
#include <iostream>
#include <fstream>
#include <memory>
#include <vector>
#include <string>
#include <regex>
#include <map>
#include <variant>
#include <algorithm>
#include <stack>

struct Node;
using NodePtr = std::shared_ptr<Node>;
using IntBoolPair = std::pair<int, bool>;

struct Node
{
	//храним или пару следующих элементов для бинарного вызова, или следущий элемент для унарного
	std::variant<std::pair<NodePtr, NodePtr>, NodePtr> arguments;
	//храним или число - значение атома, или значение операции
	std::variant<int, char> value;
};

void readInputData(std::ifstream& in, std::vector<std::string>& data)
{
	std::string tmp;
	while (std::getline(in, tmp))
	{
		if (tmp.back() == '\r')
			tmp.erase(tmp.end() - 1);
		data.push_back(tmp);
	}
}

bool checkBrackets(std::string& expression)
{
	std::stack<char> st;

	for (auto i : expression)
	{
		if (i == '(')
			st.push(i);

		if (i == ')')
		{
			if (st.empty())
				return false;
			st.pop();
		}
	}

	return st.empty();
}
```

```
bool convertToDict(std::string& dictStringValue, std::map<std::string, int>&
variablesValues)
{
	std::regex pattern("\\(\\w+ [0-9-]+\\)");
	std::smatch match;

	while (std::regex_search(dictStringValue, match, pattern) != 0)
	{
		std::string variableData(match.str());

		auto spacePos = std::find(variableData.begin(), variableData.end(), '
');
		std::string variableName(variableData.begin() + 1, spacePos);
		std::string variableValue(spacePos + 1, variableData.end() - 1);

		if (variablesValues.find(variableName) != variablesValues.end())
			return false;

		variablesValues[variableName] = std::stoi(variableValue);
		dictStringValue.erase(match.position() + dictStringValue.begin(),
match.position() + match.length() + dictStringValue.begin());
	}

	if (dictStringValue != "()")
		return false;

	return true;
}


// данная функция заменяет подстроку вида sqrt(n) на 0
// почему не на результат?
// Задача - проверить корректность, следовательно, нет необходимости считать
значение выражения
bool convertSqrtToInt(std::string& parsedString, std::map<std::string, int>&
variablesValues)
{
	std::regex pattern("sqrt\\([0-9a-zA-Z]+\\)");
	std::smatch match;

	while (std::regex_search(parsedString, match, pattern) != 0)
	{
		std::string sqrtData(match.str());
		std::string sqrtArgument(sqrtData.begin() + 5, sqrtData.end() - 1);
		try
		{
			std::stoi(sqrtArgument);
		}
		catch (std::invalid_argument&)
		{
			if (variablesValues.find(sqrtArgument) == variablesValues.end())
				return false;
		}

		*(match.position() + parsedString.begin()) = '0';
		parsedString.erase(match.position() + parsedString.begin() + 1,
match.length() + match.position() + parsedString.begin());
	}

	return true;
}

// аналогично sqrt
```

```
bool convertLogToInt(std::string& parsedString, std::map<std::string, int>&
variablesValues)
{
	std::regex pattern("log\\([0-9a-zA-Z]+, [0-9a-zA-Z]+\\)");
	std::smatch match;

	while (std::regex_search(parsedString, match, pattern) != 0)
	{
		std::string logData(match.str());

		auto commaPos = std::find(logData.begin(), logData.end(), ',');
		std::string logArgument(logData.begin() + 4, commaPos);
		std::string logBase(commaPos + 2, logData.end() - 1);

		try
		{
			int c = std::stoi(logArgument);
			if (c == 0)
				return false;
		}
		catch(std::invalid_argument&)
		{
			if (variablesValues.find(logArgument) == variablesValues.end())
				return false;
		}

		try
		{
			int c = std::stoi(logBase);
			if (c == 0)
				return false;
		}
		catch (std::invalid_argument&)
		{
			if (variablesValues.find(logBase) == variablesValues.end())
				return false;
		}

		*(match.position() + parsedString.begin()) = '0';
		parsedString.erase(match.position() + parsedString.begin() + 1,
match.length() + match.position() + parsedString.begin());
	}

	return true;
}


std::string ExtractBracketsValue(const std::string& expression, size_t* indPointer)
{
	size_t& ind = *indPointer;
	//error - переменная необходимая для обработки данного случая (...(...)...)
она позволяет получить значение
	//лежащее точно от уже найденной открывающей до корректной закрывающей скобки
	int tmp_ind = ind, error = 0;
	std::string tmp_s;

	while (1)
	{
		//запись очередного символа
		tmp_s += expression[tmp_ind];
		tmp_ind++;

		if (expression[tmp_ind] == '(')
			error++;
```

```
            if (expression[tmp_ind] == ')')
                  error--;
            if (error < 0)
                  break;
      }

      //запись )
      tmp_s += expression[tmp_ind];

      //перенос индекса за выражение в скобках дял считывания второго аргумента
      ind = tmp_ind + 1;
      return tmp_s;
}

IntBoolPair ExtractValueForListFormation(const std::string& expression, size_t*
indPointer, std::map<std::string, int>& variablesValues)
{
      std::string numberStringForm;

      size_t& ind = *indPointer;
      //считывание значения в переменную tmp
      while (ind < expression.size() && expression[ind] != ' ' && expression[ind]
!= '(' && expression[ind] != ')')
      {
            numberStringForm += expression[ind];
            ind++;
      }

      //проверка на то, что считанное значение - переменная
      if (variablesValues.find(numberStringForm) != variablesValues.end())
            return std::make_pair(variablesValues[numberStringForm], true);

      int value = -1;
      try
      {
            value = std::stoi(numberStringForm);
      }
      catch (std::invalid_argument&)
      {
            if (variablesValues.find(numberStringForm) == variablesValues.end())
                  return std::make_pair(0, false);

            value = variablesValues[numberStringForm];
      }

      return std::make_pair(value, true);
}

bool check(std::string& expression, std::map<std::string, int>& variablesValues,
NodePtr& head)
{
      if (expression[0] != '(')
            return false;

      size_t ind = 1;

      while (ind < expression.size() && expression[ind] == ' ')
            ++ind;

      if (expression[ind] != '+' && (expression[ind] != '-' && expression[ind + 1]
== ' ') && expression[ind] != '*')
            return false;

      char operation = expression[ind];
```

```
	head->value = operation;

	++ind;
	while (ind < expression.size() && expression[ind] == ' ')
		++ind;

	auto firstArg = std::make_shared<Node>();
	if (expression[ind] == '(')
	{
		std::string res = ExtractBracketsValue(expression, &ind);
		bool checkFirstArgResult = check(res, variablesValues, firstArg);
		if (checkFirstArgResult == false)
			return false;
	}
	else
	{
		auto res = ExtractValueForListFormation(expression, &ind,
variablesValues);
		if (res.second == false)
			return false;

		firstArg->value = res.first;
	}

	while (ind < expression.size() && expression[ind] == ' ')
		++ind;

	if (expression[ind] == ')')
	{
		if (operation == '-')
		{
			head->arguments = firstArg;
			return ind == expression.size() - 1;
		}
		return false;
	}

	auto secondArg = std::make_shared<Node>();
	if (expression[ind] == '(')
	{
		std::string res = ExtractBracketsValue(expression, &ind);
		bool checkSecondArgResult = check(res, variablesValues, firstArg);
		if (checkSecondArgResult == false)
			return false;
	}
	else
	{
		auto res = ExtractValueForListFormation(expression, &ind,
variablesValues);
		if (res.second == false)
			return false;

		secondArg->value = res.first;
	}

	head->arguments = std::make_pair(firstArg, secondArg);

	return ind == expression.size() - 1;
}


int main(int argc, char** argv)
{
	if (argc > 2)
```

```
	{
		std::ifstream in(argv[1]);
		if (!in)
		{
			std::cout << "Uncorrect input file\n";
			return 0;
		}
		std::ofstream out(argv[2]);
		if (!out)
		{
			std::cout << "Uncorrect output file\n";
			return 0;
		}

		std::vector<std::string> inputData;
		readInputData(in, inputData);
		if (inputData.size() % 2 != 0)
		{
			std::cout << "Input data must consist of pairs (expression and
variable list), entered data is uncorrect\n";
			return 0;
		}

		std::vector<bool> outputData;
		for (auto it = inputData.begin(); it != inputData.end(); it += 2)
		{
			std::string& expression = *it;
			std::string& dictStringValue = *std::next(it);

			std::map<std::string, int> variablesValues;
			bool dictConvertingResult = convertToDict(dictStringValue,
variablesValues);
			if (dictConvertingResult == false)
			{
				out << "Uncorrect variable list\n";
				continue;
			}

			bool bracketsCheckResult = checkBrackets(expression);
			if (bracketsCheckResult == false)
			{
				out << "Uncorrect brackets statement\n";
				continue;
			}

			bool sqrtConvertingResult = convertSqrtToInt(expression,
variablesValues);
			if (sqrtConvertingResult == false)
			{
				out << "Uncorrect sqrt argument\n";
				continue;
			}

			bool logConvertingResult = convertLogToInt(expression,
variablesValues);
			if (logConvertingResult == false)
			{
				out << "Uncorrect log argument\n";
				continue;
			}

			auto head = std::make_shared<Node>();
			bool checkExpressionResult = check(expression, variablesValues,
head);
```

```
                if (checkExpressionResult == true)
                        out << "YES\n";
                else
                        out << "NO\n";
        }
    }

    return 0;
}
```